System Resilience at Extreme Scale

White Paper

Contributors:

Professor Ricardo Bianchini, Rutgers University, Piscataway
Professor Tarek El-Ghazawi, George Washington University, Washington D.C.
Professor Armando Fox, University of California, Berkeley
Forest Godfrey, Cray, Minneapolis
Dr. Adolfy Hoisie, Los Alamos National Laboratory, Los Alamos
Professor Kathryn McKinley, University of Texas, Austin
Professor Rami Melhem, University of Pittsburgh, Pittsburgh
Professor James Plank, University of Tennessee, Knoxville
Dr. Partha Ranganathan, HP Labs, Palto Alto
Josh Simons, Sun Microsystems, Cambridge

Editor and Study Lead:

Dr. E.N. (Mootaz) Elnozahy, IBM, Austin

Prepared for Dr. William Harrod, Defense Advanced Research Project Agency (DARPA).

Executive Summary

Today, 20% or more of the computing capacity in a large high-performance computing system is wasted due to failures and recoveries. Typical MTBF is from 8 hours to 15 days. As systems increase in size to field petascale computing capability and beyond, the MTBF will go lower and more capacity will be lost.

This document analyzes the current problem in reliable systems and suggests new avenues for research in resilient systems at extreme scale. We show that central to the current problem in providing reliability is the programming model based on flat message passing using MPI. This model does not offer any failure containment, and thus a failure in one node in the system triggers an entire system failure. As systems continue to increase in size, this is not tenable.

We also analyze the new trends in technology and show that power management, the recent trend toward heterogeneous computing and the expected increase in system size all will interact negatively with resilience at the system level.

We then propose new avenues in research that have a promise of mitigating the situation. This approach is depicted in the figure below. It consists of two parts, one is to exploit existing technology trends such as the proliferation of multi-core systems and flash memory, and the expanded use of virtualization. The second part consists of opening new areas of research based on exploiting new emerging programming models, as well as system-level implementation of resilience. New directions in statistical machine learning and compiler and run-time support for resilience are also suggested.

The criterion of success in the proposed research is to reduce the computing capacity that is wasted due to failure from today's 20% to within 2% in future systems of larger size.

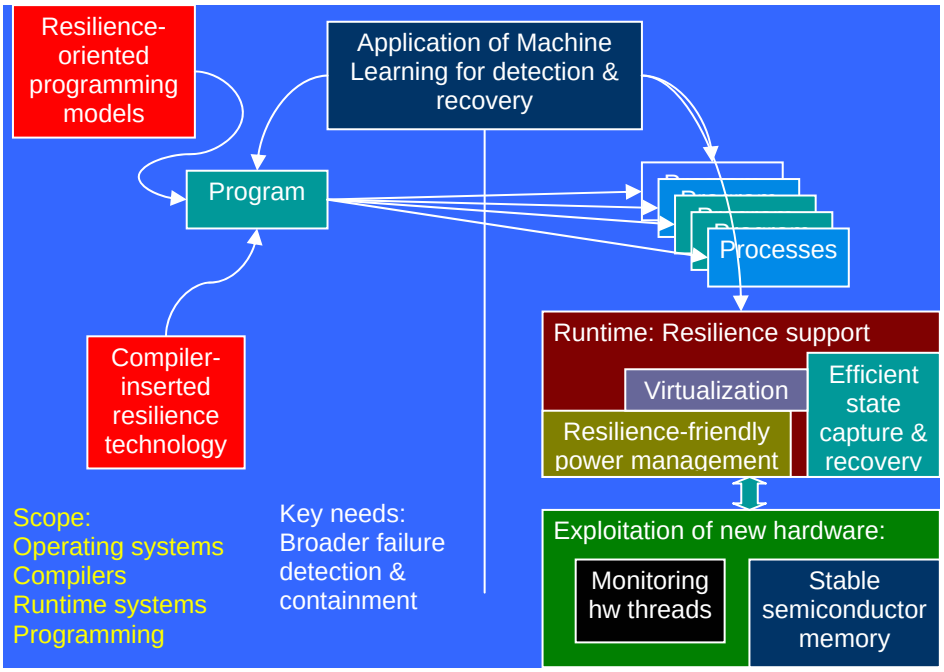

Table of Contents

1 Introduction

As systems continue to grow in size and capabilities, the current state of the art in system reliability is being pushed to its limit. This document summarizes the current and future trends in large-scale systems, with emphasis on how they affect the reliability problem. It then presents potential venues for research and innovations that exploit these trends and improve the reliability of the system in the face of current and anticipated future problems.

At the dawn of the Peta-scale computing era, it is remarkable that the current state of the art in reliability for large-scale systems has barely advanced beyond the concepts of the early 1990's. The current prevailing programming model in large-scale systems is based on message passing, with coordinated periodic checkpointing the method of choice to provide fault tolerance in practical installations. Coordinated checkpointing worked well for early ASCI machines such as ASC Blue and ASC White. But the performance and recoverability of these techniques are inadequate for modern systems that include 100,000's of cores. Notable are the performance overhead due to saving the entire system state periodically to a centralized stable storage, and the failure of these methods to protect healthy nodes from the effects of failed nodes, leading to cascading aborts throughout the system due to a single node failure. The performance overhead is due to the need to save the state of the application to stable storage made out of a centralized, slow and expensive disk storage facility. This state size is measured in Tera Bytes and soon in Peta Bytes, while disk speeds have barely improved in the past decade. A member of the team has related that current systems in his organization are typically stopped for about 10 minutes each hour to perform the checkpointing operation. This translates to about 16% performance and power waste in failure-free operation alone. In the Peta-scale era, the increase in size and the reduction of the Mean Time Between Failure (MTBF) of future systems will only push the frequency of checkpointing higher, in turn increasing the overhead in power and performance and exacerbating the pressure on the network and the stable storage devices. These projections are not tenable.

Reliability in future systems will also be affected by the system interaction with power consumption. At the current rate of 3MW/PF, multi-petaflop systems will require data centers that are capable to provide 10's of megawatts from the power grid. System vendors will have to deploy power management technologies to minimize the energy spent in a given computation, but the impact of these technologies on system reliability is not at all understood. For example, exploiting idle cycles in the system to reduce power will reduce system temperature, which in turns reduce leakage and improve immunity against soft errors in the electronic components. However, the continuous change in voltage and frequency will introduce thermal and concomitant mechanical stresses on the electronic chips and board-level electrical connections. These stresses may cause device damage and failures. Another example is power management techniques for disks. Today's enterprise-level disks that are typically used in large-scale systems are not designed to withstand many power cycles. Any power management technique therefore is bound to reduce the Mean Time to Failure of the storage subsystem. If one considers that many applications in a large-scale system have synchronized computation, file access and communication phases. The effect of employing power management in for these applications will subject the power grid in the data center to power swings measured in megawatts over as little as microseconds. It is not even clear that the current design of data centers can sustain such swings.

We also observe a trend toward integrating the data center facilities into the computing systems. For example, future large scale systems will likely use water cooling, with the water being drawn from the data center cooling subsystem. The computer management subsystem will expand beyond the traditional boundaries of the computing system to include control over the data center cooling and thermal management. Thus, new modes of failures are likely to be present as the failure in the devices that were traditionally outside the domain of the computing system will be manifested directly into system outages. Furthermore components such as board-level fans and power supplies will be stressed by the variation due to power management, and these may accelerate their failure rates. Current reliability models do not address these problems, and one must conclude then that the interactions between reliability and power is not understood, and the magnitude of the problem must be studied.

Another trend that we predict is the heterogeneity of future systems. The only Peta-scale system that exists today is composed of two heterogeneous processors. Other accelerators such as Field Programmable Arrays (FPGA), vector accelerators and special-purpose computers will be common in future systems. These accelerators may not run a conventional operating system, may not offer the usual detection of failures that we are used to in conventional architectures, and it may not be straightforward to capture their state into the system's state as part of a checkpoint, for instance. Handling the failure itself is not well understood. For example, if a failure in an FPGA occurs, do we declare the entire machine as failed, even though the "main node" remains healthy?

Even the question of what constitutes a failure will be an important field of study. Today, the reliability technology that we have handles only what is commonly referred to as "crash" failures. In reality, system crashes result from detected hardware failures that force a system check, or from operating system failures (e.g. hangs). But the systems of the future will be complex, where a computation unit or a node will consist of possibly heterogeneous multicore chips, and it is worth exploring if a partial failure in the node could be handled in a manner that would allow the node to continue operation, albeit at a reduced performance while recovery is attempted. In other words, a "crash" may not be the only type of failure that the system should detect and recover from. Partial failures must be contained and tolerated, and every effort must be made to prevent the partial failure of a node from propagating throughout the system. This also should extend the ability of the system to detect and recover from the effects of soft error beyond the traditional machine check. Soft errors occur in hardware and their rate is likely to increase because of the continuous shrinking of computer chips. It is both desirable and necessary to avoid converting such errors into machine crashes as occurs today. The system should be able to detect and partially recover from such errors, so as to contain the effect of failures over the entire system. The same applies to failures that occur to the software. With the increasing complexity of having virtualized system software and potentially multiple images of operating system within a node, it will be desirable to ensure that the state of each operating system is continuously monitored and any anomalous situations detected and fixed in a manner that does not force the application to running on that node to fail.

2 Understanding Failures

Computer system failures occur due to a variety of sources, including hardware, software and human errors. Many studies have been conducted to understand the nature of these failures, their

occurrence frequency and their impact on system's mission. These failures impact the system in different ways, depending on the nature of the solution deployed and its size.

2.1 Failures in Commercial Systems

It is useful to consider how failures are handled in the commercial domain before we delve into how they are handled in large-scale systems that are typically used for scientific computing. In the commercial domain, data processing systems have a typical mission of conducting a large number of relatively independent operations that affect a large, shared database. A typical commercial system consists of a 3-tiered structure, namely the presentation or Web layer, the application or logic layer, and the data or database layer (see **Error! Reference source not found.** [1]). This popular model has become the prevailing one in commercial applications.

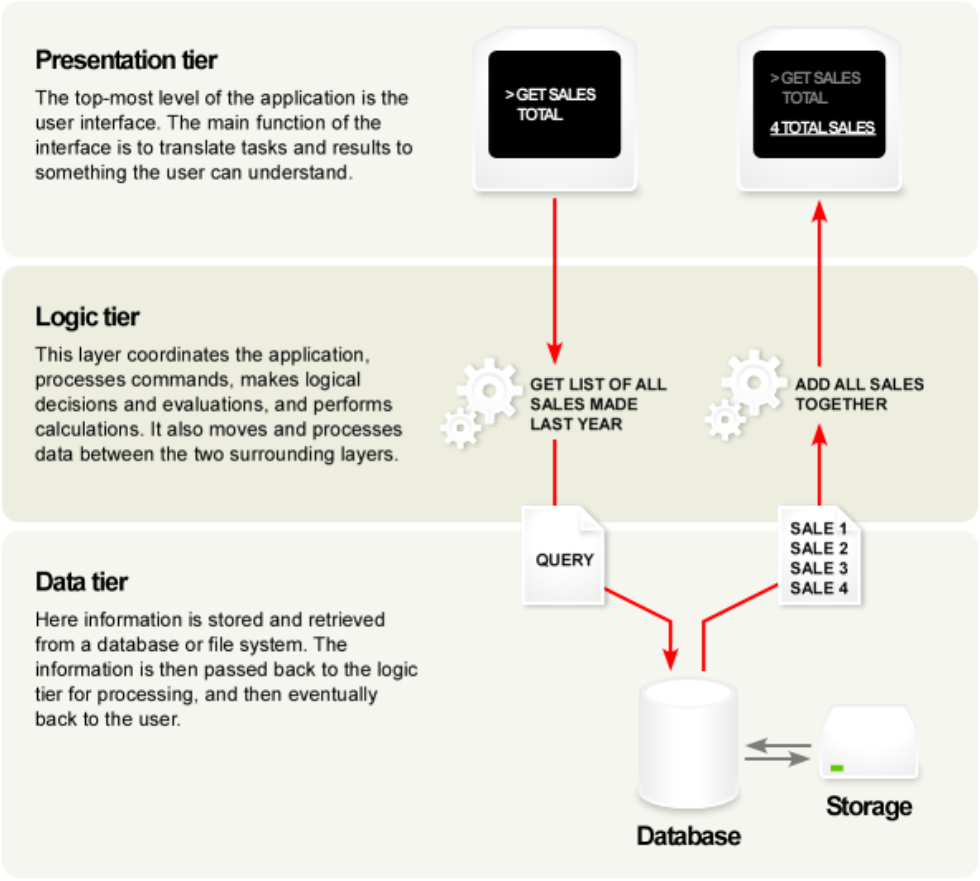

Figure 1. An example of a 3-tiered commercial application.

Hardware failures, software bugs and human errors affect commercial systems . Over the years, a certain practice has developed in which the application is structured as a collection of short running *transactions* that perform atomic updates to a replicated or data-redundant database [1]. A transaction processing system is provided to guarantee the so-called ACID properties, namely Atomicity, Consistency, Isolation and Durability. Replication and other techniques have been developed to make the database highly available in face of failures. These properties provide a

very useful abstraction to the transaction writer: A transaction takes place in an ideal failure-free environment in which the transaction appears to operate on durable data in isolation of other transactions. Failures have an all or nothing effect—either the transaction completes or it has never run, and a failed transaction never affects the database. The transaction processing system transparently manages concurrency control and failure atomicity.

Transactions enable the programmer to avoid the issues of concurrency control and failure recovery to a large degree and thus simplify application programming. And because transactions are short, their failures do not have a high cost associated with them. Of course, transaction processing impacts performance, and data redundancy consumes additional storage and data bandwidth resources. Furthermore, as more transactions attempt to manipulate shared data, the scalability of the system is tested.

Yet, despite of the costs associated with transaction processing, it has become the method of choice in commercial data processing systems. One of its particular advantages is that a failure tends to have an isolated effect on the system. A failure affects only a limited number of transactions, which can be restarted with relatively a small cost. Thus, a failure is not an impediment to scalability. In practice, we see that transactions enable commercial systems to be built at a very large scale. This is in stark contrast with scientific computing systems as will be discussed.

2.2 Failures Sources in Large-Scale Systems

Large-scale systems are relatively few and inaccessible, and therefore studies to assess their reliability and failures are rare. Recently, Schroeder and Gibson published a study on the sources of failures in two large-scale parallel systems [4]. The study surveyed over 22 production-level systems of different sizes and processor types at the Los Alamos National Laboratory. The study also included one large Non-Uniform Memory Access (NUMA) system of 512 processors per node. Various failures have been observed and tabulated. Failures included hardware failures in CPU and memory; software failures in operating systems, parallel file systems, middleware and applications; and human errors in system configuration and administration. Figure 1 shows distribution of failure sources and their respective frequencies in two different systems in the study.

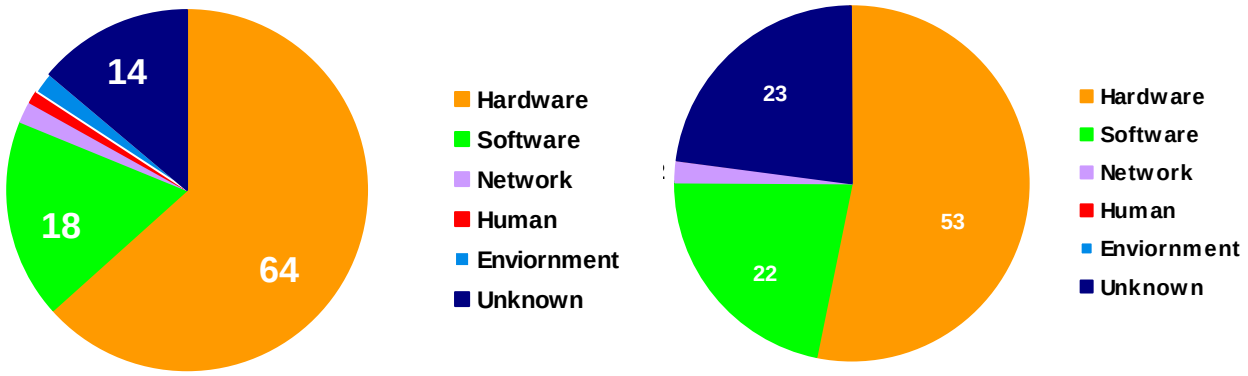

Figure 2 Failures in two sample systems.

The data show that in the two systems, hardware failures dominate the other sources. This contrasts with previous studies in commercial systems which have shown software and human errors as the dominating sources, while hardware failures tend to be limited and declining over time [1]. We believe that there are two reasons for the dominance of hardware failures in a large-scale system devoted to scientific computing. First, the software stack tends to be relatively simple. Most programs tend to be numeric intensive, rarely exercise the operating system, and the only middleware they use is the Message Passing Interface (MPI). Therefore, it is reasonable to expect that the simplicity of the software will cause relatively fewer failures, in contrast with commercial systems in which software complexity is rampant. The second reason for the dominance of hardware failures is the scale of the system. With a very large number of hardware components, the system-level probability that one of them fails will be more significant than in the smaller scale commercial systems that were studied before.

The study shows also that failures occur with a surprising frequency. It appears that compute-intensive applications stress system hardware in ways that commercial systems do not. The systems under study had Mean Time Between Failures (MTBF) ranging from eight hours to 15 days, and a Mean Time To Repair (MTTR) ranging from about one hour to one day. The large MTTR time requires that large-scale systems must be provisioned with spare compute nodes that can be brought on line immediately to compensate for the nodes that fail. Otherwise, the computation will have to run on a fewer number of nodes. Often, this is simply not possible given that many applications for examples require a number of computing nodes that is a power of 2, or have built-in load management and distribution algorithms that will break if the number of machines is reduced. Generally, application code is not written to show flexibility in face of failures, and thus the failed components must be replaced for the computation to continue. This is a common practice in large installations. This provisioning, however, is a waste of resources if no failure occurs.

2.3 Failures and Scalability

A particular problem that compute-intensive parallel systems suffer from is the lack of *failure containment.* Programs in these environments tend to deploy a large number of nodes to

implement a single computation, and use MPI with a flat model of message exchange in which any node can communicate with another. As a result, a node that participates in a computation acquires dependencies on the states of the other nodes. A failure in one node, thus, is a failure of the entire computation, since the computation cannot continue until the failed node is brought back to a state that is consistent with all the nodes in the computation. But repairing the node may require other healthy nodes to roll back their state to regenerate messages that are necessary for repair. As a result, all nodes that participate in the computation may have to roll back because one of them failed. A theory of distributed system has been developed to reason about this problem [2]. This theory states that fundamentally, message-passing systems are complex because messages induce inter-process dependencies during failure-free operation. Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called *rollback propagation*. To see why rollback propagation occurs, consider the situation where a sender of a message *m* rolls back to a state that precedes the sending of *m*. The receiver of *m* must also roll back to a state that precedes *m*'s receipt; otherwise, the states of the two processes would be *inconsistent* because they would show that message *m* was received without being sent, which is impossible in any correct failure-free execution. Under some scenarios, rollback propagation may extend back to the initial state of the computation, losing all the work performed before a failure. This situation is known as the *domino effect.*

Several solutions have been developed for the problem of recovering from a failure in a parallel systems based on message passing (such as MPI). All these solutions deploy some form of checkpointing during failure free operation, with an assortment of message logging variations that offer different tradeoffs between the performance impact of checkpointing and logging during free failure operation on one hand, and the extent of rolling back among healthy node upon a failure on the other hand. The checkpoint typically is taken to a stable storage device that is redundant and highly available. A network storage system based on disks and equipped with redundant connections to the system is typically used for the purpose of storing the checkpoints from the various nodes in the system. We note here that taking the checkpoint to a local disk within a node is not a good solution because if that node becomes disabled due to a failure, the disk is no longer accessible, and the computation cannot be restarted.

The lack of failure containment and the fact that one node failing may affect the entire computation limits the scalability of the system. If the probability of one node failing remains constant, increasing the system size simply increases the probability of a failure in one of its nodes. Ultimately, the rate of failures can be such that the computation ceases to make any progress, suffering from a repeated occurrence of failures and repairs. Given that the computations are often long-running and may exceed the time during which the system stays healthy, it follows that failures and recoveries may impede the scalability of the system. We now turn to how these issues are being solved today and the current state of the art in dealing with failures.

2.4 Current State of the Art: Checkpoint and Rollback

All available studies have shown that writing the state of a process to stable storage is the largest contributor to the performance overhead of checkpointing [2]. The simplest way to save the state of a system is to suspend execution of *all* processes at all nodes, wait for all messages that are currently in transit to reach their destinations, wait for all message queues to be emptied, then

save to stable storage the process's address space, register values, and all necessary data to reconstruct the process if necessary in the future. The execution is then resumed. This scheme can be costly for programs with large address spaces if stable storage is implemented using magnetic disks, as it is the custom.

The above mechanism is often referred to as system-level checkpointing. If a failure occurs in any node in the system, all processes are restored from the state that was last saved on stable storage. Execution then resumes from that point.

Figure 3 shows a timeline of system-level checkpointing regularly during failure-free (or normal) operation and restarting from an earlier saved state if a failure occurs.

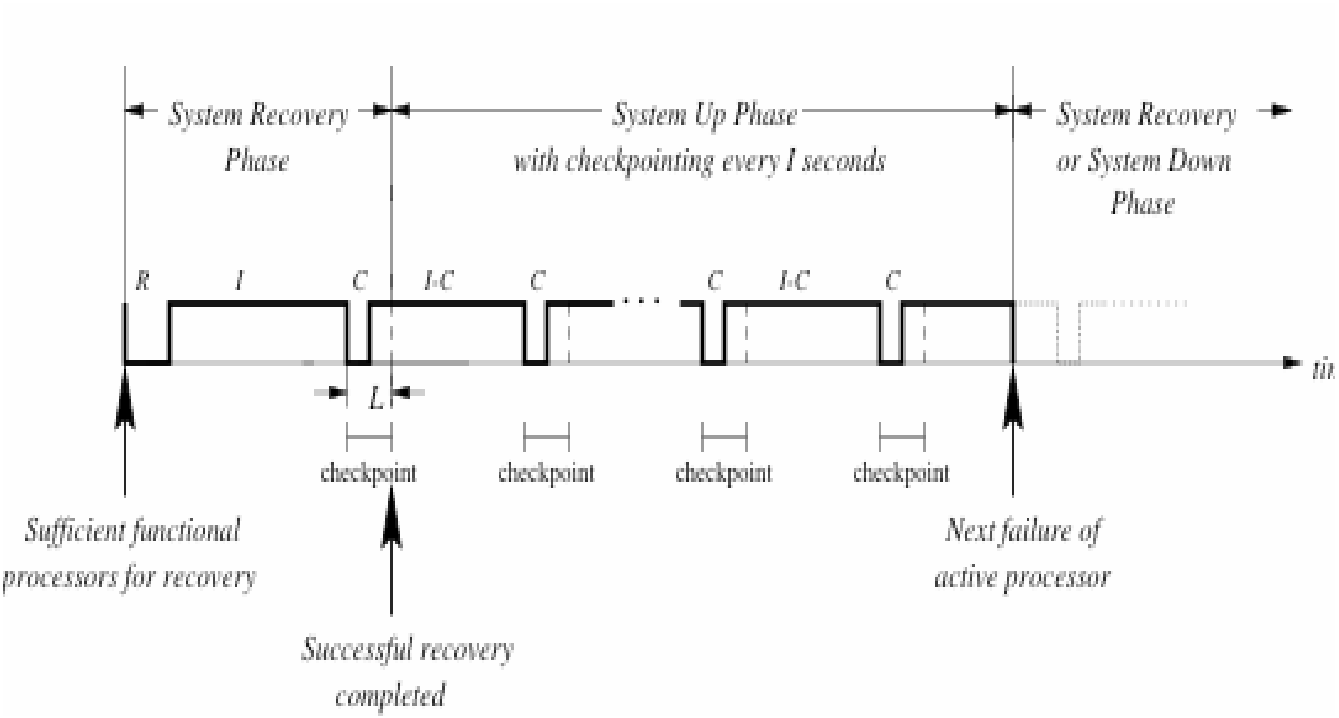

Figure 3 Failure and recovery action over time.

System-level checkpointing is expensive, and yet it is currently the state of the art in implementing checkpointing in production systems. The expense is due to stable storage access and the fact that the system stops doing useful work while the checkpoint is taken. Consider a system where the amount of main memory available on the aggregate is 1 Petabytes. It would take 1000 seconds for a sophisticated stable storage device that can store data at an effective rate of 1 Terabyte/second. This is obviously absurd, and shows that the next generation parallel systems will have to pay a tremendous premium on provisioning stable storage systems. Post petascale systems will require even more expensive resources in data bandwidths and stable storage bandwidth. Clearly, the situation is not tenable.

2.5 Checkpointing Implementations

2.5.1 Checkpointing Frequency

There is a tradeoff between the frequency of checkpointing and cost of rollback during failures. More frequent checkpoints reduce the amount of work that is at risk to be lost due to a failure. If a system takes a checkpoint every hour, then on average, about half-an-hour worth of work could be lost if a failure occurs. A large amount of work has been devoted to analyzing and deriving the optimal checkpointing frequency and placement. The problem is usually formulated as an optimization problem subject to constraints. The current practice however considers the MTBF, the amount of overhead of checkpointing and the amount of work at risk as the main driving factors in choosing the frequency of checkpointing. We explain how this is typically done by an example.

If the system is expected to have an MTBF of 1 day, then in theory the minimum number of checkpoints is twice that rate per day to ensure that the system will ultimately finish the computation. However, failures do not occur exactly according to the stated MTBF. Also, the user may not be comfortable with the notion of losing half-a-day worth of work due to a failure. Therefore, in situation of this sort, a more frequent checkpointing rate is typically desired to limit the amount of work at risk. Today, a checkpoint on the hour is the typical frequency deployed in many systems. It is also reported that for current systems of 100TF or more, a checkpoint typically requires 10 minutes to complete.

As the system scale increases, the MTBF will go down, requiring or forcing a higher frequency of checkpointing. Eventually, the system may be simply bound in taking checkpointing with very little work done if the MTBF goes down to a few hours. This situation is depicted by

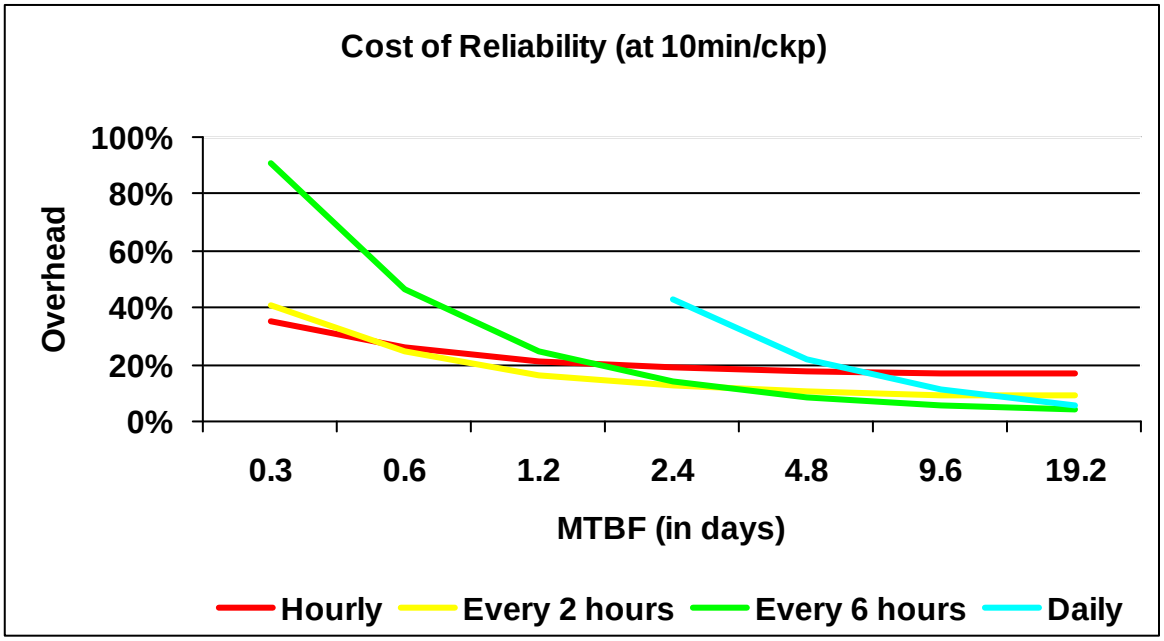

Figure 4. Cost of reliability as a function of MTBF.

Figure 3 showing how the overhead of checkpointing varies with the MTBF. The overhead in the figure takes into account the expected overhead due to failure restart. As seen from the

figure, as the MTBF drops to 8 hours, the system is expected to spend 40% of its time doing checkpointing and recovery if a checkpoint is taken on the hour. Less frequent checkpoints, say at one/6 hours, will cost the system more than 90% of its time, most likely because of the checkpointing rate leaves a lot of work at risk due to the high frequency of failures. This is not tenable, and is another aspect of how the current practice is limited in its upward scalability.

2.5.2 Possible Optimizations

Checkpointing implementation can be improved by reducing the overhead of the checkpointing process itself, specifically by reducing the amount of state that must be saved and by overlapping the execution of the application with the saving of the state. Concurrent checkpointing relies on the memory protection hardware available in modern computer systems to continue the execution of the process while its checkpoint is being saved on stable storage. The address space is protected from further modification at the start of a checkpoint and the memory pages are saved to disk concurrently with the program execution. If the program attempts to modify a page, it incurs a protection violation. The checkpointing system copies the page into a separate buffer from which it is saved on stable storage. The original page is unprotected and the application program is allowed to resume. Implementations that do not incorporate concurrent checkpointing may pay a heavy performance overhead unless the checkpointing interval is set to a large value, which in turn would increase the time for rollback.

Adding incremental checkpointing to concurrent checkpointing can further reduce the overhead. Incremental checkpointing avoids rewriting portions of the process states that do not change between consecutive checkpoints. It can be implemented by using the dirty-bit of the memory protection hardware or by emulating a dirty-bit in software.

Incremental checkpointing can also be extended over several processes. In this technique, the system saves the computed parity or some function of the memory pages that are modified across several processes**Error! Reference source not found.**. This technique is very similar to parity computation in RAID disk systems. The parity pages can be saved in volatile memory of some other processes thereby avoiding the need to access stable storage. The storage overhead of this method is very low, and it can be adjusted depending on how many failures the system is willing to tolerate.

Another technique for implementing incremental checkpointing is to directly compare the program's state with the previous checkpoint in software, and writing the difference in a new checkpoint. The required storage and computation overhead to perform such a comparison may waste the benefit of incremental checkpointing. Another variation on this technique is to use probabilistic checkpointing. The unit of checkpointing in this scheme is a memory block that is typically much smaller than a memory page. Changes to a memory block are detected by computing a signature and comparing it to the corresponding signature in the previous checkpoint. Probabilistic checkpointing is portable, and has lower storage and computation requirements than required by comparing the checkpoints directly. On the downside, computing a signature to detect changes opens the door for aliasing. This problem occurs when the computed signature does not differ from the corresponding one in the previous checkpoint, even though the associated memory block has changed. In such a situation, the memory block is excluded from the new checkpoint, which therefore becomes erroneous. A probabilistic analysis has shown that the likelihood of aliasing in practice is negligible, but an experimental evaluation has shown that probabilistic checkpointing could be unsafe in practice.

A compiler can be instrumented to generate code that supports checkpointing. The compiled program contains code that decides when and what to checkpoint. The advantage of this technique is that the compiler can decide on the variables that must be saved, therefore avoiding unnecessary data. For example, dead variables within a program are not saved in a checkpoint though they have been modified. Furthermore, the compiler may decide the points during program execution where the amount of state to be saved is small. Despite these promising advantages, there are difficulties with this approach. It is generally undecidable to find the point in program execution most suitable to take a checkpoint. There are, however, several heuristics that can be used. The programmer can provide hints to the compiler about where checkpoints should be inserted or what data variables should be stored. The compiler may also be trained by running the application in an iterative manner and by observing its behavior. The observed behavior could help decide the execution points where it would be appropriate to insert checkpoints. Compiler support could also be simplified in languages that support automatic garbage collection. The execution point after each major garbage collection provides a convenient place to take a checkpoint at a minimum cost.

2.5.3 Current State of the Art

Checkpointing implementations available to production systems tend to be fragile. Operating System-level implementations have been available on some commercial systems, but they tend to be accompanied with a long list of caveats that make it difficult for the programmer to assert whether it can be safe to deploy it for a particular program. Open-source and public-domain implementations that operate at the user-level have also been available, but since they cannot capture the state of the system, they may not be sufficiently robust to handle all applications. Frustrated with this state of affair, most programmers have taken matters into their hands, implementing routines to support application checkpointing specifically designed for their programs. The advantage of this approach is that it allows the programmer to decide what to save and when to save it. Yet, this is also the disadvantage of this method. A mistake by the programmer in implementing the checkpointing process may make it impossible for the application to restart correctly if a critical variable was not saved, or if the checkpointing is not done frequently enough. Also, it requires the programmer to complicate the logic of the program with checkpointing, and understand the MTBF terms of the system, hardly a recipe for portability or robustness. The performance overhead in this user-level checkpointing is also high, for instance, because the programmer has to use the file system interface to write the checkpoints and cannot rely on any system-level optimization of the storage structure and since everything has to be done from the program level, additional layers of overhead make the process more complex and expensive.

3 Future Trends and Impact on Resilience

Several trends in technology are emerging and will affect the construction of future systems. We explore some of the trends and how they will affect system resilience. The purpose of this exploration is to understand how incremental improvement in resilience will be effective in face of these technology trends, and identify areas where investments need to be directed to prepare future systems.

3.1 Power Management

For environmental as well as economical reasons, power consumption of computing systems is a major concern for system designers and users. A full survey of the state of the technology in power management is not in the scope of this report. What is relevant here is how power management will interact with system resilience.

Current and future power management technologies will encompass all system components, including processor cores, memories, storage, I/O circuitry, power supplies, service processors, etc. These technologies include slowing components down, such as Dynamic Voltage and Frequency Scaling (DVFS) of processor and memory chips, slowing down the rotational speed of magnetic disks, and speed control of data communication on external network fiber or network switches. More aggressive power management techniques will also include resource deactivation such as turning off individual cores within a multi-core processor chip, disabling some cache lines and turning off specific memory banks. Turning off disks has been used in mobile systems for years and is finding its way into large-scale server environments. We will also see power supplied to a specific subsystems being capped.

Power management will be performed voluntarily under application control or involuntarily under operating system or even hardware control. For example, power management can be instigated by programmer control by providing hints to the operating system about the parts of the program where processor speed can be reduced (e.g. while waiting for a message). But the programmer may not have ultimate control, for example the hardware may decide to slow the processor down to reduce thermal stresses and hot spots under extreme conditions of workload intensity.

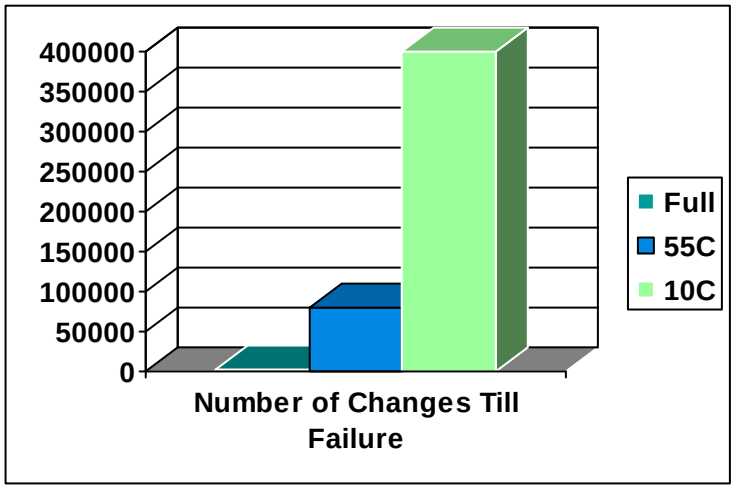

Figure 5 Effects of continuous power on and off of a chip on its longevity.

The interaction of power management with system resilience is likely to be negative. Power management will create thermal variations that will induce mechanical stresses at chip and board levels. Furthermore, the continuous change in the disk rotational speed or the frequent activation and deactivation of disks will create unprecedented reliability challenges to the mechanical components of the disks. For example, enterprise-class disks are designed to sustain a power on-off every 8 hours, certainly less frequent than power management systems are expected to

change the disk speeds. The mechanical stresses, when accumulated over time, may lead to board-level failures in the form of separated or shorted connections. Additionally, the accumulated effects of mechanical and thermal stress will reduce the longevity of individual chips and magnetic disks. Figure 5 shows the expected number of turn on-off cycles as a function of the thermal swing. Notice the order of magnitude reduction in chip longevity when the thermal swing increases from 10C to 55C. The net effect of all the failure acceleration may be the reduction of the expected reliability of the individual components (component Mean Time To Failure, MTTF), which in turn may negatively affect the system's overall MTBF. Alternatively, it may also cause more expensive qualification and components testing, which may drive a large system cost beyond the realm of affordability.

At the large scale, there may be some electrical stresses at the data center level. For example, if power management is applied to reduce the voltage and power of 100,000's of processors simultaneously will create large power swings of megawatts within a few microseconds. We note today that large-scale installations have a routine by which machine bring up and down are cascaded through out the system to avoid these swings.

Another area where our understanding is incomplete is the interaction of power management with soft errors. There will be two opposing effects at play due to the operation at lower voltages. On one hand, the threshold for errors will be reduced and thus tolerance to soft errors at the system level will be reduced. On the other hand, lower power consumption will reduce temperature and white noise, which will counter act the first effect, potentially. Models need to be developed to study the effects of these interactions.

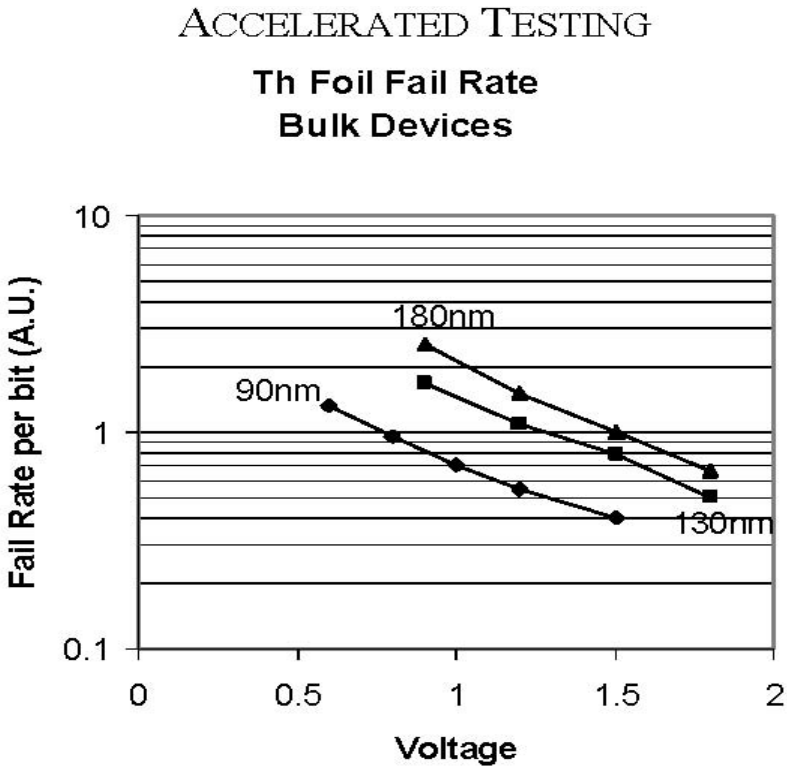

Figure 6 Variation of SER with voltage for 3 Silicon generations.

Power management also may interact in interesting ways with existing software. So far, software has been designed and tested with all components running at the same speed. It is conceivable that with power management changing the speeds of these components, some timing bugs may be uncovered. It is reasonable to argue that software should not have any bugs, but the reality is that functioning software often contains latent bugs that could get exposed if power management starts interacting with the system in ways that were not tested before. At the very least, software testing has to include regression tests that mitigate these effects.

The interactions of power management with resilience, thus, are conjectured to be negative, and we emphasize that no one currently has sufficient experience to assess these interactions and their impact. Also, it is clear that resilience-aware power management algorithms need to be developed at all levels of software and hardware design. We therefore believe that studying the effect of applying power management on the reliability of the system should be an area of research worthy of pursuit.

3.2 System Size

Processor speed improvement due to technology is essentially over. Going forward, Moore's law will allow only denser integration of transistors, which means that systems will grow horizontally. Also, disk speeds are not likely to improve much. Performance demands will force designers to a brute force approach in which more performance will be met by incorporating more components in a system. We already know that the Blue Gene super computer includes about 128,000 processor cores in its maximal configuration. We also know that HPCS-class machines will likely include 100,000's of processor cores, and thus, beyond petascale systems could possibly include millions of processors, memory chips and disks. Assuming the traditional improvement in component-level reliability, the rapid increase in the number of system components will likely reduce the MTBF of the system to a few days or even hours. As discussed in Section 2, the lower MTBF may well force existing techniques for reliability out of consideration. For example, an MTBF of 8 hours may yield an unacceptable overhead of 40% at 10 minutes/checkpoint at the system level. Also, preserving the checkpoint duration may require a non linear increase in the number of disks that are devoted to save the checkpoints, merely to keep up with the checkpoint demands. This may not be even affordable from a financial standpoint. For example, today's system balances are typically 0.5B/F for memory capacity and 0.01 B/F for memory bandwidth. For these balances, the checkpointing time would be about a minute, ideally. System inefficiencies tend to push this figure higher. At any case, the balance appears impossible to sustain at much higher system compute capability. Therefore, it is clear that the current practice *must* be revised at the envisioned level of system sizes for the petascale systems and beyond. New techniques for resilience, possibly impacting the existing practice of writing large-scale programs must be developed.

3.3 Heterogeneity

The studies performed under the HPCS program and others have pointed out the serious limitations of the current model of writing parallel programs based on MPI. Besides its scalability limits, we have pointed out in Section 2 that it has an inherent problem with failure containment, which will likely limit the scalability even further. Recently, we have seen a new trend toward incorporating heterogeneous processors in the system in the form of accelerators,

Field Programmable Gate Arrays (FPGA), and processors of different types. An example of these new systems is the Road Runner system at the Los Alamos National Laboratory.

It is not clear at this point how one can produce a portable technology of providing reliability for these heterogeneous systems. For the current practice of checkpoint/restart, for instance, will require support in capturing the states of the various components. This is not straightforward, for instance, the state of an FPGA may not be captured easily, and some additional support for capturing a meaningful state of the system is needed. Also, application software on these systems will be more complex than the versions that are targeting a single, homogeneous platform. Therefore, it is conceivable that the reliability of the software (operating system, middleware and applications) will reduce the MTBF of the system further. We believe that further research must be directed at developing new algorithms and protocols for building reliable heterogeneous systems. The current status quo is inadequate and will not field the new challenges on the horizon.

4 Plausible Fields of Research

In the preceding sections, we have established that the current practice in building resilient systems will not carry through with the larger systems envisioned at petascale and beyond. New research is needed to develop the necessary technology that will achieve the goal of ensuring acceptable levels of resilience in future systems. In this section, we study potential fields and directions of innovations toward this goal. Some of these directions are inspired by new technology trends that can be exploited to improve system resilience. Others are insights that we have gained by observing what happens in other fields.

4.1 Exploiting New Technology Trends

4.1.1 Exploiting Multi-Core Systems

Future systems will feature processors with many cores. Limitations on memory bandwidth, cache capacity and application software may prevent some of these cores from being put to profitable use all the time. A simple idea is to dedicate some of this processing capability to handle recovery chores and support an increased level of system reliability. This approach seeks to create a different value out of the additional cores other than mere performance.

As an example to illustrate the point, consider the study shown in Figure 7, depicting the performance of four applications on a 4-threaded core. It is clear that there is very little performance improvement that can result from increasing the number of threads from 1 to 2, much less from 2 to 4. In all but one commercial application (JBB), performance hardly improves.

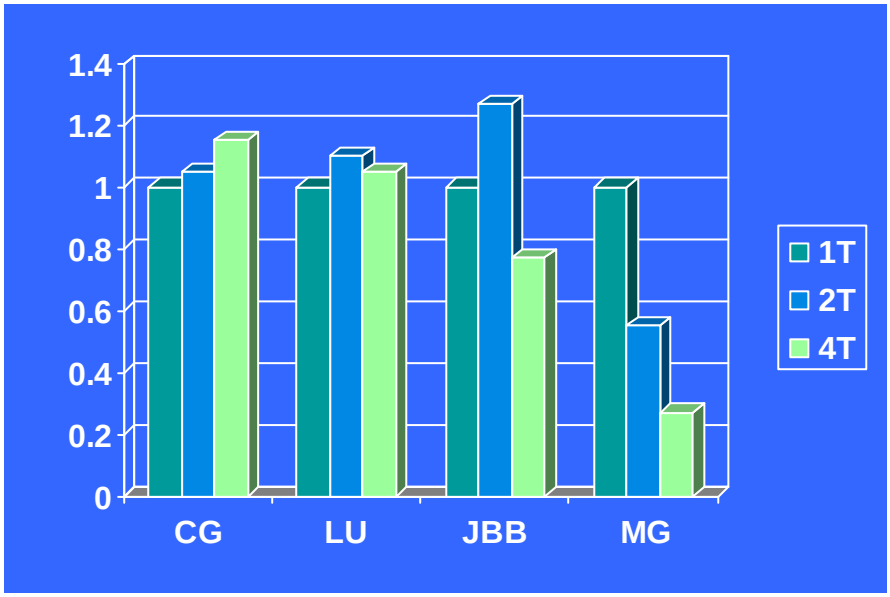

Figure 7 Performance study on a 4-threaded core performance**.**

Exploiting multi-core systems can take the form of allocating one core (or more) to perform all the monitoring logic that is required by higher level algorithms for providing reliability. For example, Figure 8 shows an example of an 8-core processor chip, in which 2 cores are set aside for resilience support. This support may include diagnosis and error checking, or could be made available to the compiler to include reliability support. How to do this requires a lot more research, but it appears a plausible and promising approach.

Figure 9 shows another potential use of multi-core systems, in which the cores are paired in the modules, each consisting of a primary and a backup. This approach could be used to detect and recover from soft errors, and could yield substantial redesign of the processor logic system. For example, this design allows less emphasis on diagnostic and error checking circuitry in the logic, which could reduce the design complexity and power consumption. Again, how to do this requires further research both in hardware and software (e.g. the operating system must be involved). The approach again appears plausible and promising.

A research agenda can be built around exploiting multicore systems for reliability purposes, including further investigations of the examples provided here and others. There is a need to prototype and explore these possibilities to determine the most effective exploitation of additional cores and threads. System software will also be impacted and it may not be trivial to include such support (e.g. compiler exploitation of multicore for the purpose of reliability). We also need to assess the interactions with power management, and how it affects resilience in such configurations. Finally, this has to be integrated into an overall strategy for providing resilience in future systems.

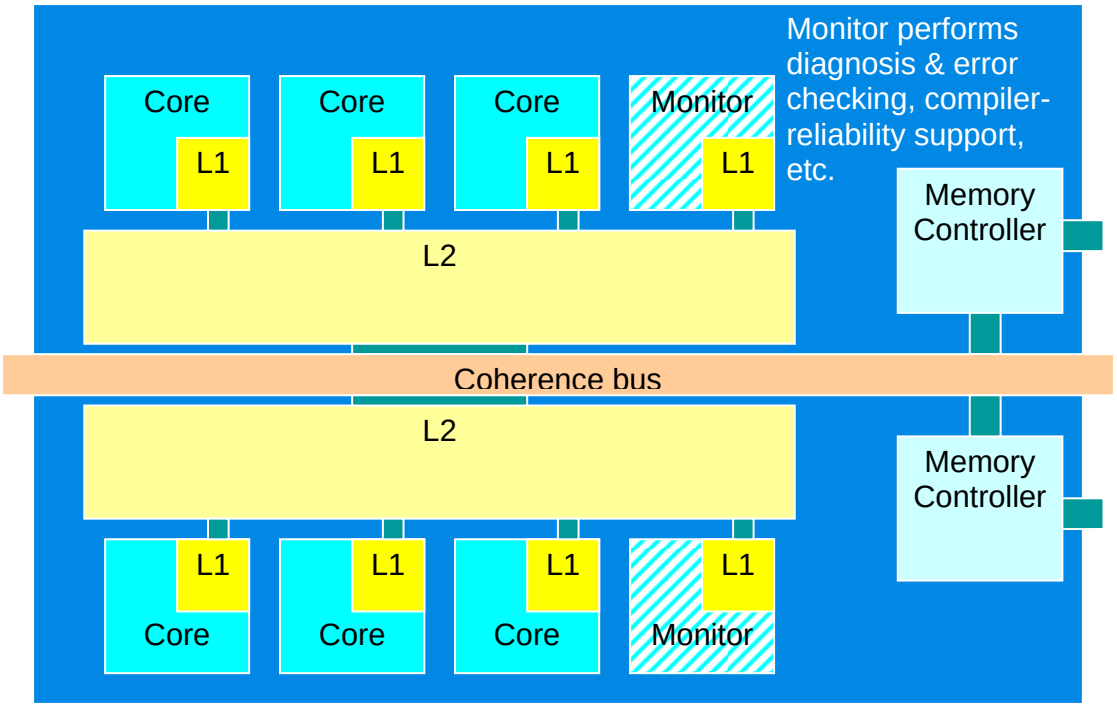

Figure 8 A multicore system with two cores devoted to resilience support.

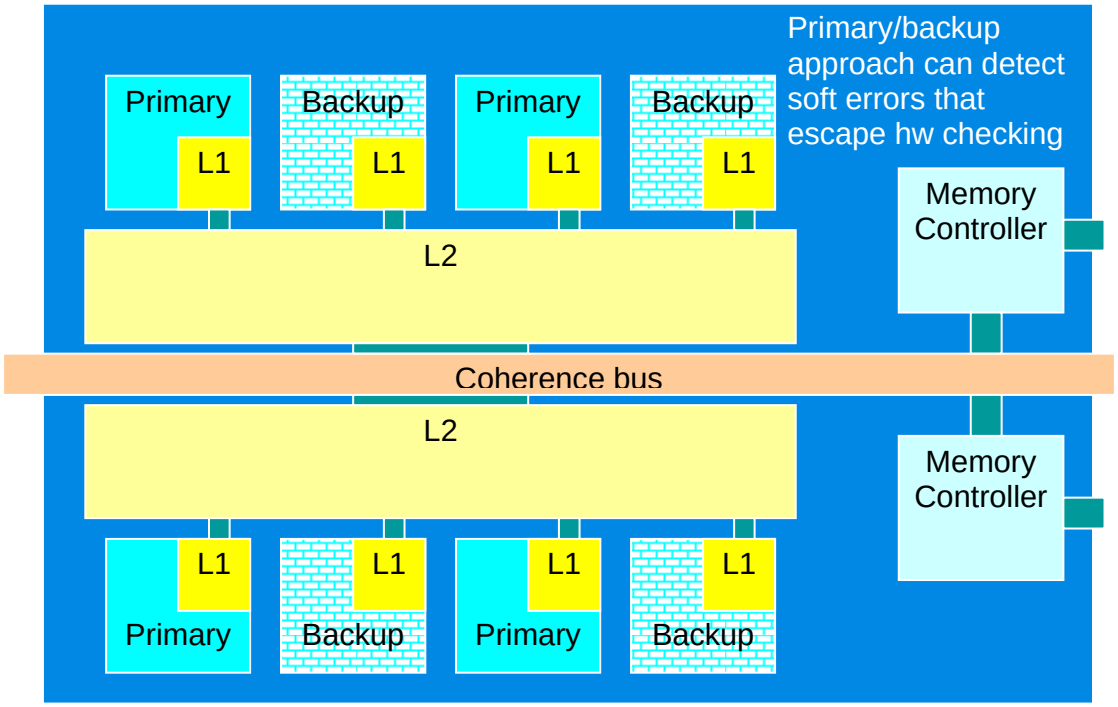

Figure 9 A multicore system with a primary/backup approach.

4.1.2 Exploiting Semiconductor-based Stable Storage

As described in Section 2, a central stable storage facility is required to support resilience and reliability in today's systems. We argued that this approach, however, is not scalable, especially when we consider that future systems may have lower MTBF and therefore a requirement for more frequent checkpointing. We also know that the focus on strong scaling with require at least a 10X reduction in checkpoint latency. These requirements cannot be met by incremental improvement of existing checkpointing techniques and disk-based stable storage. A new trend in technology is the availability of semiconductor-based stable storage, e.g. in the form of flash memory.

Flash memory can be exploited to extend the life of existing checkpointing techniques. For instance, it can be used as a local flash disk to store the checkpoints within each system. Then, when all checkpoints have been captured, which would be much faster than today's approach, the checkpoints are staggered to the disk in a manner that reduce the overall contention on the disks. This may allow existing disk-based checkpointing to continue to be useful while coping with the requirements of increasing the checkpointing scheme. This 2-level approach is shown in Figure 10.

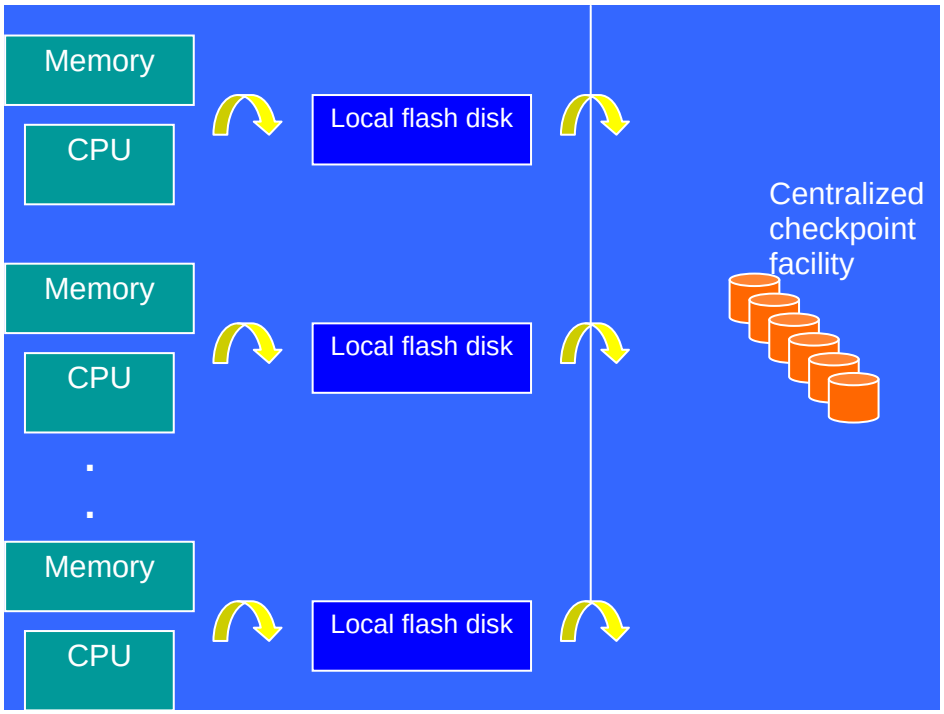

Figure 10 A 2-level checkpointing approach that exploits flash memory.

Another approach is to eliminate disk-based checkpointing altogether, and use the flash memory in a neighboring system instead. Figure 11 shows this proposal. The exploitation of flash memory is certainly a vast area of research, and we have only shown two proposals. Many others are possible and certainly this is one of the most promising approaches going forward.

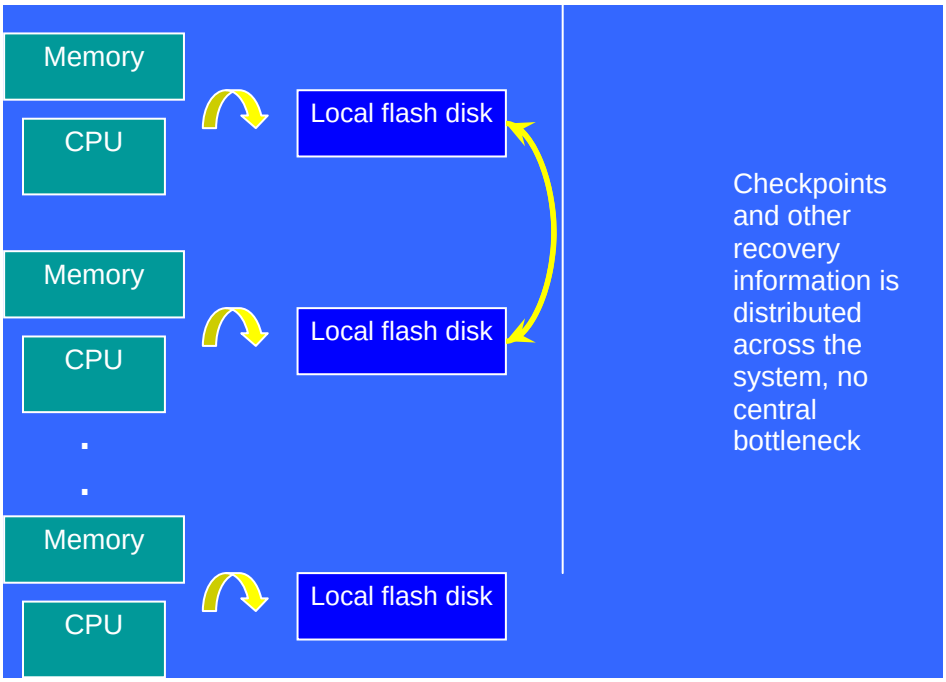

Figure 11 Distributed checkpointing using flash memory.

4.1.3 Exploiting Virtualization

Virtualization is gaining momentum in the commercial domain. Virtualization can be leveraged in high-performance computing systems as it provides a robust state capture and migration. This can be used as a robust building block to implement failure recovery, for example through checkpointing an entire partition. This relieves the application programmer from having to manage reliability and closes the gap in robustness between existing fragile checkpointing techniques and the desired level of robustness.

4.2 New Technologies

4.2.1 New Programming Models

We have established that the flat message passing model based on MPI is at the root of the failure containment problem, which we perceive as the most severe going into petascale systems and beyond. Other programming style are emerging (or re-emerging). For example, what used to be called the “bag of tasks model” in the 1990’s is back under the name of map-reduce. Other models such as the one employed in X10 confines the communication into semantically controlled interactions that could be constrained for the purpose of reliability. These programming models do not suffer from the legacy of MPI, and therefore it is useful to re-examine the current practice of checkpoint restart as the main method to provide resilience in such systems.

Some of these new programming models is inherently fault-tolerant, and thus is more scalable. For example in Figure 12 shows how the map-reduced model can help failure containment. In this model, the tasks communicate through data repository such as a histogram or input files. As a result, if one of the processing nodes fails, it can be restarted on any other system, unlike in MPI system when a single failure requires the entire system to be restarted.

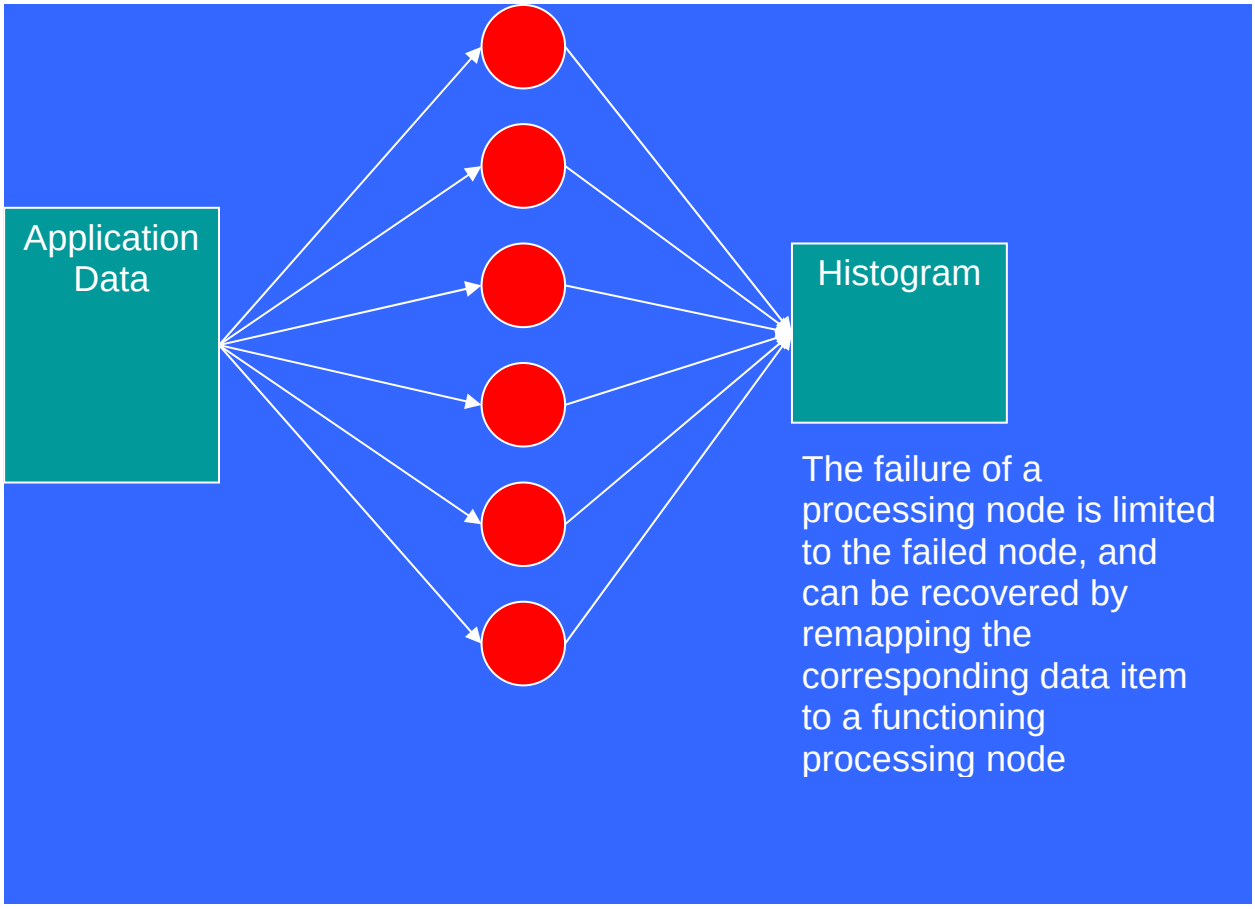

Figure 12 Containment in the map-reduce model.

4.3 Application of Machine Learning

Machine learning techniques use monitoring to build a finger print of the normal system behavior such that over time, any deviation from this normal behavior can be detected and handled. For example, machine learning techniques can be used to sift through the massive amount of data in system logs, which are intended for system administrators but are never read. Also, over time, these statistical machine learning techniques can build fingerprints for different pathological system behaviors. A database of fingerprints can thus be built over time and analyzed continuously to detect and potentially correct (or help in correcting) errors.

4.4 Compiler Support for Resilience

Software remains an important source of failures in high-performance computing systems. Currently, software verification and exhaustive testing are still beyond the capability of today's systems. Also, there are environment-dependent bugs such as memory exhaustion timing anomalies that cause problems that are not detected during normal operation. Bug detection logic inserted into the application by the compiler is an open area of research that has so far not been tapped.

A potential structure of the solution would add modules to automatically patch software while it is operational, modules for error detection and tolerance, including for instance detection of memory leaks, races, and monitoring techniques for anomaly monitoring. Coupled with the abundance of cores that we described in Section 4.1.1, there is a new promising field of research on how to equip code to cope with all sorts of failures in a programmer-transparent manner.

5 Bibliography

1. K. Birman. "How and Why Computer Systems Fail," in *Reliable Distributed Systems, Springer New York,* pp. 237—246.
2. E.N. Elnozahy, L. Alvisi, Y-M. Wang, and D. Johnson. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems." In *ACM Computing Surveys,* vol. 34, Sep 2002.
3. J. Gray and A. Reuter. "Transaction Processing: Concepts and Techniques", Morgan Kaufmann, 1993.
4. B. Schroeder and G. Gibson. "A Large Scale Study of Failures in High-Performance Computing Systems", in Proceedings of the *International Symposium on Dependable Systems and Networks* (DSN2006).
5. Wikipedia. "Multitier Architecture", the Wikimedia Foundation.